# Mathematics for Decision Making: An Introduction

## Lecture 11

Matthias Köppe

UC Davis, Mathematics

February 10, 2009

# Dijkstra's Algorithm [1959]

## Dijkstra's Algorithm

**Input:** A digraph $G = (V, A)$ with nonnegative arc costs, starting node $r$
**Output:** A predecessor vector $\mathbf{p}$, encoding minimum-cost paths from $r$ to all nodes.

1. Initialize $\mathbf{y}$, $\mathbf{p}$.
2. Set $S := V$.
3. While $S \neq \varnothing$:
   Choose $v \in S$ with $y_v$ minimum.
   Set $S := S \setminus \{v\}$.
   Scan vertex $v$, i.e., do for all arcs $(v, w) \in A$:
      If $(v, w)$ is incorrect, then correct it, updating predecessor information.

# Dijkstra's Algorithm: Correctness

- We use the notation $v_1, v_2, \ldots, v_n$ for the ordering of the nodes
- We denote by $\mathbf{y}^{(i)}$ the value of $\mathbf{y}$ at the point when $v_i$ is chosen to be scanned.

## Lemma (Monotonicity of potentials of scanned nodes)

*For all $i < k$ we have $y^{(i)}_{v_i} \leq y^{(k)}_{v_k}$.*

## Proof.

- Suppose the contrary, i.e., there exist $i < k$ with $y^{(i)}_{v_i} > y^{(k)}_{v_k}$.
- Fix such a $i$ and choose $k$ minimal with this property, i.e., $v_k$ is **the earliest-chosen vertex after** $v_i$ that, at the time of its scanning, had a smaller potential than the vertex $v_i$ at the time of its scanning.
- But by the minimal choice in the algorithm, we have $y^{(i)}_{v_i} \leq y^{(i)}_{v_k}$.
- So $y_{v_k}$ must have been lowered while scanning some vertex $v_j$ with $i < j < k$.
- This arc correction made $y^{(k)}_{v_k} = y^{(j+1)}_{v_k} = y^{(j)}_{v_j} + c_{v_j,v_k}$.
- Because $c_{v_j,v_k} \geq 0$, we have $y^{(j)}_{v_j} \leq y^{(k)}_{v_k} < y^{(i)}_{v_i}$.
- This is a contradiction to the definition of $k$. □

# Dijkstra's Algorithm: Correctness, II

## Theorem

*Dijkstra's Algorithm is correct.*

## Proof.

We prove that, after all vertices have been scanned, we have a feasible potential $\mathbf{y}^{n+1}$:

- Suppose not, i.e., for some $(v_i, v_k) \in A$, we have $y_{v_i}^{(n+1)} + c_{v_i,v_k} < y_{v_k}^{(n+1)}$.
- But directly after scanning vertex $v_i$, we certainly did have $y_{v_i}^{(i+1)} + c_{v_i,v_k} \geq y_{v_k}^{(i+1)}$.
- Since we never increase the potentials, $y_{v_i}$ must have been lowered afterwards! Say, it was lowered the last time when scanning vertex $v_j$ (with $i < j$).
- Thus $y_{v_i}^{(i+1)} > y_{v_i}^{(n+1)} = y_{v_i}^{(j+1)} = y_{v_j}^{(j)} + c_{v_j,v_i} \geq y_{v_j}^{(j)}$
- On the other hand, by the Lemma, because $v_j$ was scanned after $v_i$, we have $y_{v_j}^{(j)} \geq y_{v_i}^{(i)}$, a contradiction ($y_{v_i}^{(i+1)} > y_{v_i}^{(i)}$). □

# Dijkstra's Algorithm: Efficiency

## Theorem (Efficiency of Dijkstra's Algorithm)

*Dijkstra's Algorithm terminates after $m = |A|$ arc verification steps.*

- Let's try out Dijkstra's Algorithm in practice; we expect that the running time essentially only depends, linearly, on the number of arcs.
- We try on examples with the same number of arcs, but different numbers of vertices.
- Result: There is a great dependence on the number of vertices, and we are **not happy** with the running time for large, sparse graphs (many vertices, few arcs)
- Where is the running time spent? **Our coarse abstraction of running time (number of arc verification steps) does not give the answer.**
- To find this out in the practical program, **it is strongly recommended to find this out by measuring time, rather than thinking or guessing.**
- Every modern, reasonable programming system has a facility for measuring how much running time is spent in parts of the program; this is called a **(time) profiler**.
- In the case of C, the GCC toolchain (compiler/linker option `-pg`) and the `gprof` tool provide a (sampling) time profiler.

# Dijkstra’s Algorithm: Efficiency, II

- To make refined mathematical statements about the running time of Dijkstra’s Algorithm, we analyze the algorithm on an abstraction of a computer, which we call the **Random Access Machine** (RAM).
- Such a machine has a fixed (immutable) **program**, a **central processing unit** with finitely many **registers**, and direct (indexed by a constant) and indirect (indexed by the contents of a register) access to infinitely many **memory locations**.
- Each of the registers and memory locations can store an **integer of arbitrary size**.
- The running time of a program on the RAM is the number of **elementary operations** it executes.
  - Reading a number from memory into a register
  - Writing a number from a register to memory
  - Elementary arithmetic operations ($+$, $-$, $\times$, division with remainder) on registers
  - Comparing numbers ($=$, $\leq$, $\geq$) in registers
  - Elementary control flow operations (branches)
- In other words, by definition, each of the above elementary operations takes constant time (1 time unit). Note that this is a dramatic simplification of the running time of a program on a real computer.

# Dijkstra's Algorithm: Efficiency, III

- We now turn to the refined analysis of Dijkstra's Algorithm, based on a concrete **implementation** of the algorithm on a RAM:
  - We need to clarify how the input data are presented
  - We need to decide using which **concrete data structures** we store the data
  - We need to clarify several steps of the algorithm

  (The same is necessary if we want to create an implementation of the algorithm in a not-too-high-level programming language such as C.)
- We will assume that the digraph $(V, A)$ is given in the form of an **adjacency list**, stored in **arrays** (i.e., using contiguous memory locations), which allows to
  - obtain the number of vertices in constant time $c_1$
  - given a vertex index $v$, to determine the **outdegree** $\delta^+(v)$ (the number of arcs leaving $v$) in constant time $c_2$
  - given a vertex index $v$ and an index $i$, to determine the endpoint $w$ of the $i$-th arc leaving $v$, and the arc cost $c_{v,w}$ in constant time $c_3$
- We will store the potential vector **y** and the predecessor vector **p** as arrays. Accessing (reading or writing) an element $y_v$ or $p(v)$ of these vectors, given a vertex index $v$, then takes a constant $c_4$ many elementary operations
- We will store the set $S$ as a **singly-linked list**; this allows to decide whether $S = \varnothing$ in time $c_5$, iterate through the elements in time $c_6$ (per element), add an element at the front in constant time $c_7$, and delete an element found by iterating in constant time $c_8$.

# Dijkstra's Algorithm: Efficiency, IV

We now determine the precise number of elementary operations.

- We use the constants $c_i$ associated with the data structures, which appeared to the previous slide.
- We use additional constants $d_i$ to denote the number of elementary operations in other parts of the program.

## Dijkstra's Algorithm

**Input:** A digraph $G = (V, A)$ with nonnegative arc costs, starting node $r$
**Output:** A predecessor vector $\mathbf{p}$, encoding minimum-cost paths from $r$ to all nodes.

1. Initialize $\mathbf{y}$, $\mathbf{p}$ — $c_1 + d_1 + |V|(2c_4 + d_2)$ operations
2. Set $S := V$. — $d_4 + |V|(c_7 + d_3)$ operations
3. While $S \neq \emptyset$: — $|V|$ iterations and $(c_5 + d_5)(|V| + 1)$ operations
   - Choose $v \in S$ with $y_v$ minimum. — $d_6 + |S|(c_4 + c_6 + d_7)$ operations
   - Set $S := S \setminus \{v\}$. — $c_8$ operations
   - For all arcs $(v, w) \in A$: — $\delta^+(v)$ iterations, $c_2 + \delta^+(v)c_3$ operations
     - If $y_v + c(v, w) \leq y_w$: — $2c_4 + d_8$ operations
       - $y_w := y_v + c(v, w)$ — $c_4$ operations
       - $p(w) := v$ — $c_4$ operations

# Dijkstra's Algorithm: Efficiency, V

Adding up everything:

- The minimum-finding operation takes $d_6 + |S|(c_4 + c_6 + d_7)$ operations, where $|S|$ starts with $|V|$ and is decreased until it reaches 1. Thus its total time is:

$$\sum_{s=1}^{|V|} \big(d_6 + |S|(c_4 + c_6 + d_7)\big) = |V| d_6 + \frac{|V|(|V|+1)}{2}(c_4 + c_6 + d_7)$$

- All node-scanning operations (verifying all outgoing arcs) together take

$$\sum_{v \in V} \big(c_2 + \delta^+(v)(c_3 + 4c_4 + d_8)\big) = |V| c_2 + |A|(c_3 + 4c_4 + d_8)$$

- The remaining operations are easy to account for
- Together we obtain

$$e_1 |V|^2 + e_2 |V| + e_3 |A| + e_4$$

  elementary operations, for some (complicated) constants $e_i$.
- **For sparse graphs, where $|A| \ll |V|^2$, the term $e_1 |V|^2$ is the largest summand.** It comes from the minimum-finding operation!

# Dijkstra's Algorithm: Efficiency, VI

- **We are not happy with the complicated analysis** (counting of operations, lots of constants, ...) we had to do to obtain this result.
- Moreover, the constants $e_i$ we obtained still depend on the specific RAM we are using. For instance, on a version of a RAM with few registers, we might need more elementary operations to do the same thing.
- For these reasons, it is useful and convenient to **ignore the specific constants** and just ask **how does the running time grow for large problems** (i.e., asymptotically)
- We will use the **Landau notation** for asymptotic growth. Fix a function $g(n) \geq 0$.
  - A function $f(n) \geq 0$ is said to **grow (asymptotically) at most with order** $g(n)$ if
  $$\exists c > 0, n_0 \in \mathbf{N} : \forall n \geq n_0 : f(n) \leq cg(n).$$
  We use the notation $f(n) \in \mathrm{O}(g(n))$, this is read as "big oh of $g(n)$".
  - A function $f(n) \geq 0$ is said to **grow (asymptotically) at least with order** $g(n)$ if
  $$\exists c > 0, n_0 \in \mathbf{N} : \forall n \geq n_0 : f(n) \geq cg(n).$$
  We use the notation $f(n) \in \Omega(g(n))$, this is read as "big omega of $g(n)$".
  - A function $f(n) \geq 0$ is said to **grow (asymptotically) with order** $g(n)$ if $f(n) \in \mathrm{O}(g(n))$ and $f(n) \in \Omega(g(n))$ (note: different constants are allowed); we write $f(n) \in \Theta(g(n))$ (read: "big theta of $g(n)$")
  - Similarly, for functions of several arguments.

# Dijkstra's Algorithm: Efficiency, VII

- Using Big-Oh notation, we obtain that the running time of our RAM implementation of Dijkstra's Algorithm is

$$\Theta(|V|^2).$$

  In particular, the number of arcs (and thus sparsity) is no longer visible.
- A Big-Oh calculus helps to simplify the expressions:
  - For example, any polynomial function $p(n) = \sum_{i=0}^{d} p_i n^i$ (with $p_d \neq 0$) is in $\Theta(n^d)$.
  - In particular, constants get consumed by higher-order terms
  - $\max\{f_1(n), f_2(n)\} \in \mathrm{O}(f_1(n) + f_2(n))$
- By keeping in mind that we are only interested in this kind of asymptotic estimate, we can simplify our counting of elementary operations: We can be "sloppy", in a controlled way.
  - It suffice to determine that some operation is $\mathrm{O}(1)$, or $\Theta(n)$; we don't need to discuss the precise number of iterations.

# Dijkstra's Algorithm: Efficiency, VIII

- We are **still not happy** with the performance of Dijkstra's Algorithm for large, sparse graphs
- We have found the reason: Running time is (asymptotically) dominated by the minimum-finding operation.
- A solution is to use **better concrete data structures**. Here it pays off to use a **binary heap** (an implementation of a **priority queue**) to implement the set $S$ together with the potential vector $\mathbf{y}$.
- A priority queue stores elements $v$ together with a **priority** $y_v$; it has **operations**:
  - Empty?
  - Insert and element $v$ with priority $y_v$
  - Find, remove, and return the element $v$ of smallest priority $y_v$
  - Find a given element $v$, and change its priority to $y'_v$.
- The binary heap implementation of this abstract data structure on a RAM has running time of $\mathrm{O}(\log n)$ for all of these operations, where $n$ is the number of elements stored.